Lecture 12: Memory Management: The Kernel, Address Spaces, and Protection

12.0 Main Points:

What is an address space?
How is it implemented?

Physical memory	Abstraction: virtual memory
No protection	Each program isolated from all others and from the OS
Limited size	Illusion of infinite memory
Sharing visible to programs	Transparent -- can't tell if memory is shared
Easy to share data between programs	Ability to share code, data

12.1 Motivation for protection

What we've done so far: link program with operating system. PCs work this way: just load program into low memory, OS in high memory (each program can assume it starts at location 0).

Problem: If program crashes, operating system may be corrupted. In worst case, program might overwrite disk!

Goal of **protection**: Firewalls to
Keep user programs from crashing OS
Keep user programs from crashing each other

12.2 Hardware support for protection

How is protection implemented?

Hardware support:

address translation

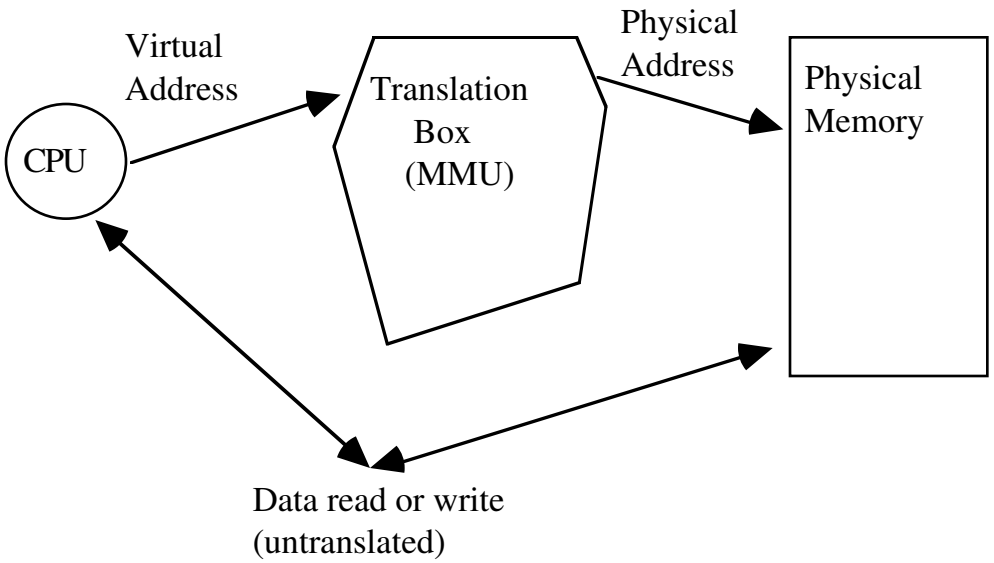
dual mode operation

12.3 Address translation

Address space: literally, all the addresses a program can touch. All the state that a program can affect or be affected by.

Restrict what a program can do by restricting what it can touch!

Hardware translates every memory reference from virtual addresses to phyiscal addresses; software sets up and manages the mapping in the translation box.



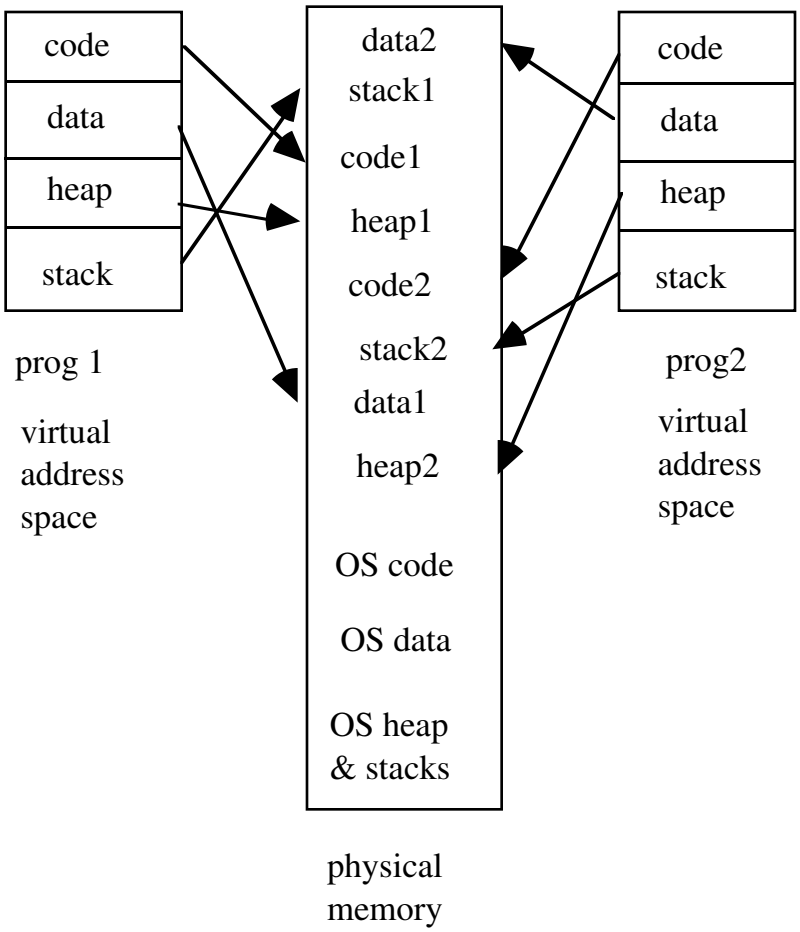
Address Translation in Modern Architectures

Two views of memory:

view from the CPU -- what program sees, virtual memory

view from memory -- physical memory

Translation box converts between the two views.



Example of Address Translation

Translation helps implement protection because no way for program to even talk about other program's addresses; no way for them to touch operating system code or data.

Translation can be implemented in any number of ways -- typically, by some form of table lookup (we'll discuss various options for implementing the translation box later). Separate table for each user address space.

12.4 Dual mode operation

Can application modify its own translation tables? If it could, could get access to all of physical memory. Has to be restricted somehow.

Dual-mode operation

when in the OS, can do anything (kernel-mode)

when in a user program, restricted to only touching that program's memory (user-mode)

Hardware requires CPU to be in **kernel-mode** to modify address translation tables.

This is how Nachos works (and most UNIXes and other non-PC OS's):

OS runs in kernel mode (untranslated)

User programs run in user mode (translated)

Want to isolate each address space so its behavior can't do any harm, except to itself.

A couple issues:

1. How do programs interact? (Deal with this later.)
2. Share CPU between kernel and user programs

How does this work?

Kernel -> user

User -> Kernel

12.4.1 Kernel -> user:

To run a user program, create a thread to:

allocate and initialize address space control block

read program off disk and store in memory

allocate and initialize translation table (point to program memory)

run program (or to return to user level after calling the kernel):

set machine registers
set hardware pointer to translation table
set processor status word (user vs. kernel)
jump to start of program

12.4.2 User-> kernel:

How does the user program get back into the kernel?

Voluntarily user->kernel: **System call** -- special instruction to jump to a specific operating system handler. Just like doing a procedure call into the operating system kernel -- program asks OS kernel, please do something on procedure's behalf.

Can the user program call any routine in the OS? No. Just specific ones the OS says is ok. Always start running handler at same place, otherwise, problems!

Involuntarily user->kernel: **Hardware interrupt**, also **program exception**

Examples of program exceptions:

bus error
segmentation fault
page fault (important for providing illusion of infinite memory)

On system call, interrupt, or exception: hardware atomically

sets processor status to kernel
changes execution stack to kernel
saves current program counter
jumps to handler in kernel
handler saves previous state of any registers it uses

Context switching between programs: same as with threads, except now also save and restore pointer to translation table. To resume a program, re-load registers, change PSL, and jump to old PC.

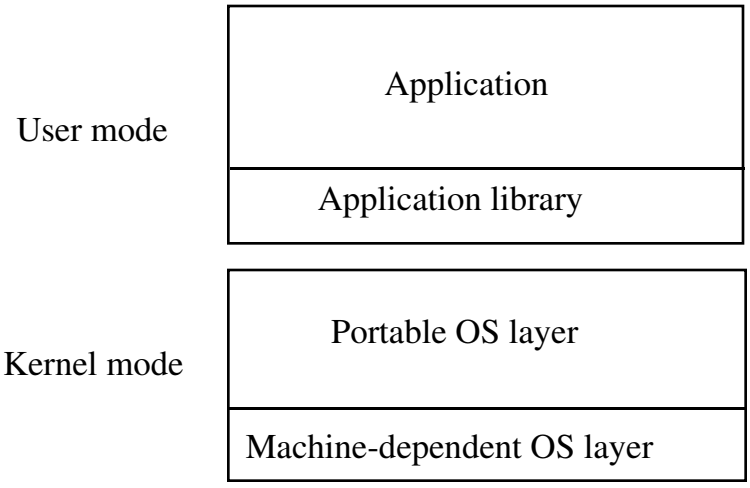
How does the system call pass arguments?

a. Use registers
b. Write into user memory, kernel copies into its memory

Except: user addresses -- translated
kernel addresses -- untranslated

Addresses the kernel sees are not the same addresses as what the user sees! Solving this riddle is a large part of assignment 2.

12.5 Operating system structure



Typical UNIX Operating System Structure

How does Nachos fit in? Nachos is the portable OS layer. It simulates the hardware and machine-dependent layer (interrupts, etc.), and it simulates the execution of user

programs running on top. (In Nachos, to run a user program, you call simulator, instead of jumping to the correct PC.)

Why do it this way, instead of building an OS on a real machine?

1. Could run Nachos on real hardware -- simply replace simulation with real hardware, and write machine dependent layer!

2. Can still run normal UNIX programs

3. Want debugger, printf, multiple users, and simpler debug cycle (5 second turnaround vs. 10 minute turnaround)

4. This is done in real life: isolate from hardware flakiness -- make sure it works before you try it on the hardware

12.6 An Example of Application-Kernel Interaction: Shells and UNIX fork

Shell -- user program (not part of the kernel!)
prompts users to type command
does system call to run command

In Nachos, system call to run command is simply "exec". But UNIX works a bit differently than Nachos.

UNIX idea: separate notion of fork vs. exec

fork -- create a new process, exact copy of current one
exec -- change current process to run different program

To run a program in UNIX:

fork a process
in child, exec program
in parent, wait for child to finish

UNIX fork:

stop current process
create exact copy
put on ready list
resume original

Original has code/data/stack. Copy has exactly the same thing!

Only difference between child and parent is: UNIX changes one register in child before resume.

Child process:

Exec program
Stop process
Copy new program over current one
resume at location 0

Justification was to allow I/O (pipes, redirection, etc.), to be set up between fork and exec. Child can access shell's data structures to see whether there is any I/O redirection, then sets it up before exec.

Nachos simply combines UNIX fork and exec into one operation.